AN3307
Application note

Guidelines for obtaining IEC60335 Class B certification in any STM32F1xx application

Introduction

The role of safety has become very important for electronics applications. The level of safety requirements for components used in electronic designs is steadily increasing. The manufacturers of electronic devices include many new technical solutions in the design of new components. Software techniques for improving safety are continuously being developed.

The current safety recommendations and requirements are specified at worldwide level by recognized international standards bodies such as IEC and come under the compliance, verification and certification process of testing houses and authorities like VDE. The certification process is closely associated with EMC tests when the robustness of the system against noise emission and noise sensitivity is tested for compliance with international standards.

The main purpose of this application note and its associated software is to facilitate and accelerate user software development and certification processes for appliances which are subject to these requirements and certifications and are based on the STM32F1xx family of microcontrollers.

The package is based on STM32F10x CMSIS and STM32F10x Standard peripheral library V3.3.0 published by ST.

The associated firmware package is available upon request only. Please contact your local ST sales office for more information.

Two projects have been prepared for IAR/EWARM version 5.41 and KEIL/MDK-ARM version 4.10 environment and tool chains.

The class B parameters support the following devices in the STM32F10x family:

- Low, medium and high density **Performance line** devices
- Low, medium and high density **Access line** devices
- Low and medium density **Value Line** devices

For more EMC information please refer to the following related application notes:

- **AN1015** Software techniques for improving EMC
- **AN1709** EMC design guide

Contents

1 **Compliance with IEC and VDE standards 6**

1.1 Generic tests included in the STM32F1xx firmware library 7

1.2 Application-specific tests not included in the ST firmware library 9

2 **Class B software package 10**

2.1 Basic software principles used 10

2.1.1 Fail Safe mode 10

2.1.2 Class B variables 10

2.1.3 Class B flow control 11

2.2 Package organization 12

2.2.1 Projects included in the package 12

2.2.2 Tool-specific integration of the library 13

2.2.3 Application demo example 13

2.3 Package configuration and debugging 14

2.3.1 Configuration control 14

2.3.2 Verbose diagnostic mode 15

2.3.3 Debugging the package 16

3 **Class B solution structure 17**

3.1 Integrating the software into the user application 17

3.2 Description of startup self tests 18

3.2.1 CPU startup self-test 19

3.2.2 Watch dog startup self-test 19

3.2.3 Flash complete checksum self-test 20

3.2.4 Full RAM March C-/X self-test 21

3.2.5 Clock startup self-test 23

3.3 Periodic runtime self-test initialization 24

3.4 Description of periodic runtime self-tests 25

3.4.1 Runtime self-test structure 25

3.4.2 CPU light runtime self-test 26

3.4.3 Stack boundaries runtime test 27

3.4.4 Clock runtime self-test 27

3.4.5 Partial Flash CRC runtime self-test 28

3.4.6 Watchdog service in runtime test 29
3.4.7 Partial RAM runtime self-test 29

Appendix A Library reference 31

4 Revision history 32

List of tables

Table 1. MCU parts which must be tested for Class B compliance 7
Table 2. Overview of methods used in the micro-specific tests provided with this application note . . 8
Table 3. Physical order of RAM addresses organized into blocks of 16 words 21
Table 4. March C- phases in RAM partial test 30
Table 5. List of self-test library files 31
Table 6. Revision history 32

List of figures

Figure 1. Example of RAM memory configuration . . . 11
Figure 2. Control flow four-step checking principle . . . 12
Figure 3. STM32 demo hyper terminal output window in verbose mode . . . 15
Figure 4. Integration of startup and periodic runtime self-tests into the application . . . 17
Figure 5. Startup self tests structure . . . 18
Figure 6. CPU startup self-test structure . . . 19
Figure 7. Watchdog startup self-test structure . . . 20
Figure 8. Flash startup self-test structure . . . 21
Figure 9. RAM startup self-test structure . . . 22
Figure 10. Scramble test - detailed structure of a single check and fill procedure . . . 22
Figure 11. Clock startup self-test subroutine structure . . . 23
Figure 12. Clock frequency measurement principle . . . 24
Figure 13. Periodic runtime self-test initialization structure . . . 25
Figure 14. Periodic runtime self-test and timebase interrupt service structure . . . 26
Figure 15. CPU light runtime self-test structure . . . 27
Figure 16. Stack overflow runtime test structure . . . 27
Figure 17. Clock runtime self-test structure . . . 28
Figure 18. Partial Flash CRC runtime self-test structure . . . 28
Figure 19. Partial RAM runtime self-test structure . . . 29
Figure 20. Fault coupling principle used at partial RAM runtime self-test . . . 30

1 Compliance with IEC and VDE standards

The IEC (International Electro technical Commission) is a non-profit and non-governmental authority recognized worldwide for preparing and publishing international standards for a vast range of electrical, electronic and related technologies. IEC standards are focused mainly on safety and performance, the environment, electrical energy efficiency and its renewable capabilities. The IEC cooperates closely with ISO (International Organization for Standardization) and ITU (International Telecommunication Union). Their standards define not only the recommendations for hardware, but also for software solutions. Standards are divided into a number of safety classes depending on the purpose of the application.

The other bodies that are recognized worldwide in the field of electronic standard organizations are VDE in Germany, IET in the United Kingdom and the IEEE in the United States. The VDE association also includes a Testing and Certification Institute which is a pioneer of software safety inspection. This is a registered National Certification Body (NCB) for Germany. The main purpose of this testing house is to offer standards compliance and quality testing services to manufacturers of electrical appliances.

One of the pivotal IEC standards is the IEC 60335-1 norm, which covers safety and security of household electronic appliances destined for domestic and similar environment. Appliances incorporating electronic circuits are subject to component failure tests.

The basic principle is that the appliance must remain safe in the case of any component failure. The microcontroller is an electronic component just like any other from this point of view. If safety relies on an electronic component, it must remain safe after two consecutive faults. This means the appliance must stay safe with one hardware failure and with the microcontroller not operating (under reset or not operating properly).

If the safety depends on software, the software is taken into account with the second applied failure. The conditions required for software are defined precisely in Annex Q of the IEC 60335-1 norm. Three classes of appliances are defined here:

- **Class A**: Safety does not rely on software
- **Class B**: Software prevents unsafe operation
- **Class C**: Software is intended to prevent special hazards

This application note and the associated ST software package covers the group B specification. Appliances under group C need some other special requirements such as dual microcontroller operation, which is outside the scope of this document.

Class B compliance aspects for microcontrollers are related both to hardware and software. A list of microcontroller parts under compliance is evaluated at IEC 60335-1 Annex T which refers to IEC60730 Annex H. This list can be divided into two groups - micro-specific and application-specific items. See *Table 1*.

While application-specific parts rely on customer application structure and must be defined and developed by users (communication, I/O control, interrupts, analog inputs and outputs), micro-specific parts are related purely to the micro structure and can be made generic (core self-diagnostic, volatile and non-volatile memory integrity checking, clock system tests). This group of micro-specific tests is the focus of the ST solution based on powerful hardware features of STM32 MCU such as dual independent watchdogs or clock sources.

Table 1. MCU parts which must be tested for Class B compliance

Group	Components to be tested
Micro-specific	CPU registers
	CPU program counter
	Clock
	Volatile & non-volatile memories
	Internal addressing (and external memory addressing if any)
	Internal data path
Application-specific	Interrupt handling
	External communication
	Timing
	I/O peripherals
	Analog A/D and D/A
	Analog multiplexer

1.1 Generic tests included in the STM32F1xx firmware library

The certified by VDE STM32F1xx firmware library packages are composed of the following micro-specific software modules:

- CPU register test
- Clock monitoring
- RAM functional check
- Flash checksum integrity check
- Watchdog self-test
- Stack overflow monitoring

An overview of the methods used for these MCU-specific tests is given in *Table 2* and they are described in more detail in the following chapters. The last two items are not explicitly asked for by the norm, but they improve overall fault coverage.

Table 2. Overview of methods used in the micro-specific tests provided with this application note

Components to be verified	Method used
CPU registers	Functional test of all registers and flags including R13 (stack pointer), R14 (link register) and PSP (Process stack pointer) is done at startup. In the runtime test R13, R14, PSP and flags are not tested. The stack pointer is tested for underflow and overflow, the link register is tested by PC monitoring. If any error is found, the software jumps directly to the Fail Safe routine.
Program counter	Two different watchdogs driven by two independent clock sources can reset the device when the program counter is lost. The Window watchdog (driven by the main oscillator) performs time slot monitoring and the Independent watchdog (driven by the low speed internal RC oscillator) must be serviced at regular intervals. Program control flow is additionally monitored using a specific software method.
Addressing and data path	This is tested indirectly by RAM functional and Flash integrity tests, stack overflow (a specific pattern is written at a low boundary of stack space and checked for corruption at regular intervals) and underflow (a second pattern is written at a high boundary if it is not at the RAM end). In addition, a bus error exception vector is fetched by the CPU if a memory access fault occurs.
Clock	A reciprocal method of comparing two independent clock sources is used while clocking two timers. Wrong main frequency (harmonic/sub harmonic) can be detected using the external low speed 32kHz oscillator as a timebase.
Non-volatile memory	A 16-bit CRC software checksum test of the entire memory is done at startup and a partial memory test is repeated at runtime (block by block). Optionally the built in hardware for fast 32-bit CRC calculation can be used.
Volatile memory	A March C- (or optionally March X) full memory test is done at startup and a partial memory test is repeated at runtime (block by block), scrambled order of physical addresses in RAM is respected in tests for optimal coverage of coupling faults, word protection with double redundancy (inverse values stored in non adjacent memory space) is used for safety critical Class B variables, Class A variable space, stack and unused space are not tested at runtime.

Users can include a part or all of these software modules certified by VDE into the project. If they stay unchanged and are integrated in accordance with the guidelines, the time and costs needed for a certified end-application would be significantly reduced.

Note: If minor changes have to be made to the modules, it is suggested that you keep careful track of all of these changes in order to inform the certification authority of all modifications made to the certified routines.

1.2 Application-specific tests not included in the ST firmware library

Users have to focus on all the remaining tests that cover application-specific MCU parts that are not included in the ST firmware library:

- Testing the analog blocks (AD/DA, multiplexer)
- Testing the digital I/O
- External Addressing
- External communication
- Timing and interrupts

The tests for the analog part depend on the application and peripheral capability of the device. The pins used should be checked for correct analog intervals (by checking that the measured voltages correspond to the real analog values) and free analog pins can be used to read some reference voltages in conjunction with testing the analog multiplexers if they are used in the application. The internal reference voltage should also be checked. Some STM32 devices feature two (and some even three) independent ADC blocks. It make sense to perform conversions on the same channel using two different ADC blocks for security reasons.

Class B tests must also detect any malfunction of the digital I/Os. This could be covered by plausibility checks together with some other application parts (for example, the change in an analog signal from a temperature sensor should be checked when the heating/cooling digital control is switched on/off). Selected port bits can be locked by applying the correct lock sequence to the lock bit in the GPIOx_LCKR register to prevent unexpected changes to the port configuration. Reconfiguration is only possible at the next reset sequence in this case. In addition, the bit banding feature can be used for atomic manipulation of the SRAM and peripheral registers.

Application interrupt occurrences and external communication flows should also be checked. Different methods can be used: one method can use a set of incremental counters with specific counters incremented by each interrupt or communication event. The values in the counters are then checked periodically at given time intervals to make a cross-check with an independent timebase. The number of events performed within the last period should correspond to the application requirements. The configuration lock feature can be used to secure the timer register settings with three levels controlled by the TIMx_BDTR register.

2 Class B software package

This section highlights the basic common principles used in the ST software solution. The workspace organization is described as well as configuration and debugging capabilities. Any differences between the two supported development environments (IAR's EWARM and Keil's MDK-ARM) are highlighted.

2.1 Basic software principles used

The basic software methods and common principles used for all the tests included in the ST solution are described in detail in this section.

2.1.1 Fail Safe mode

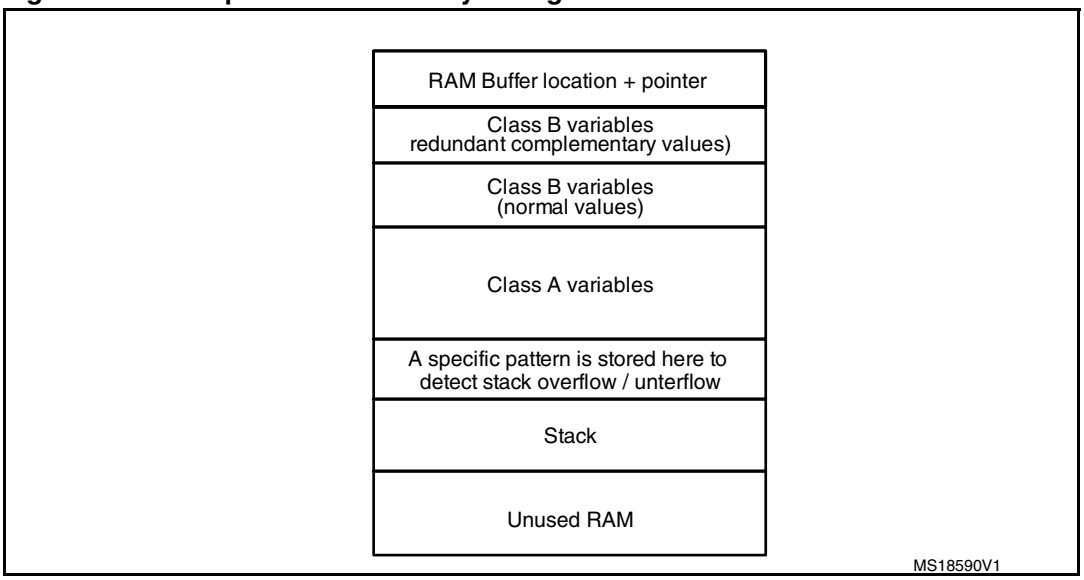
If any failure is detected, the ***FailSafePOR()*** routine is called (defined in ***stm32f10x_STLstartup.c*** file). The program stays in an endless loop waiting for a watchdog reset. Apart from some debugging features the routine is almost empty. Users have to provide the content to the routine and ensure that it executes the right actions to keep the application in a safe state.

The debug or verbose mode described in *Section 2.3: Package configuration and debugging* can be used to identify which error has occurred.

2.1.2 Class B variables

Each class B variable is stored as a pair of two complementary values in two separate RAM regions. Both normal and redundant complementary values are always placed in non adjacent memory locations. A partial transparent RAM March C- or March X test is performed continuously on these RAM areas by means of an interrupt subroutine. The pair is compared for integrity each time before the value is used. If any value stored in the pair is corrupted, Fail Safe mode is invoked. An example of RAM configuration can be seen in *Figure 1*. Users can adapt the RAM space allocation according to the application needs and the hardware capabilities of the device.

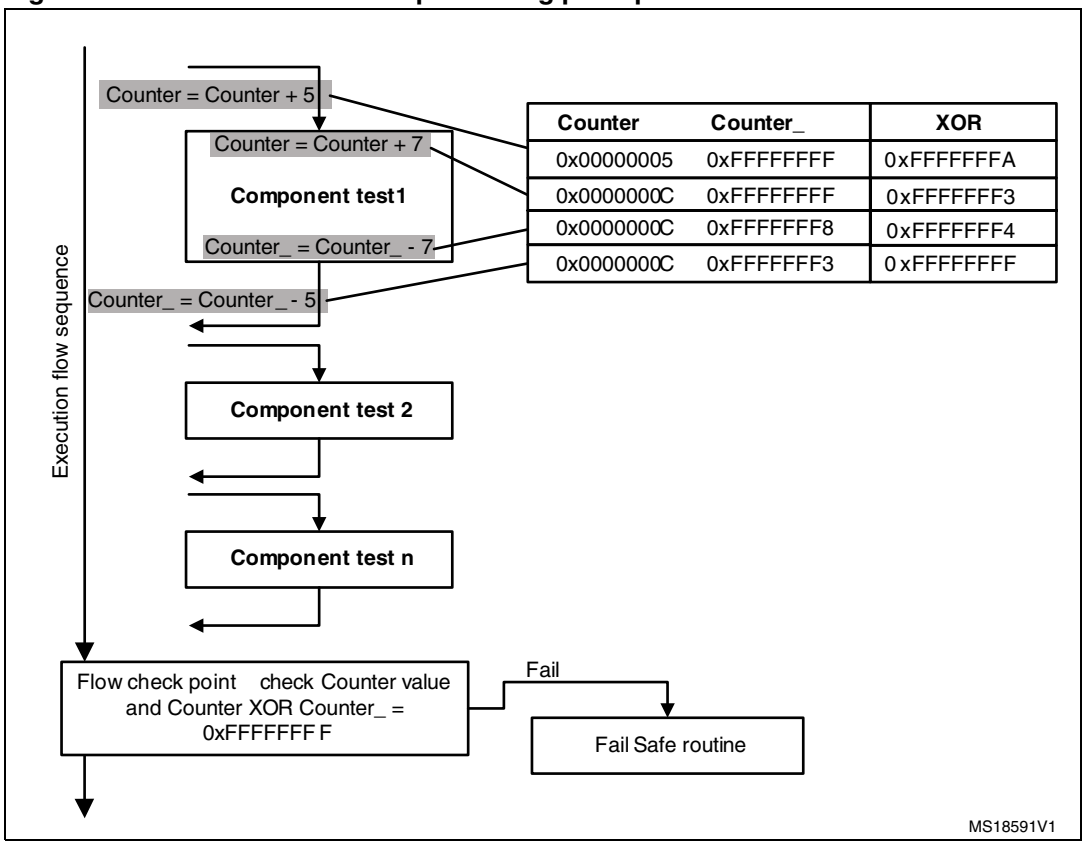
Figure 1. Example of RAM memory configuration



2.1.3 Class B flow control

A specific software method is used to check if all parts of the test flow are successfully passed. Unique labels (constant numbers) are defined for identifying all key points (blocks with component tests) in the code flow in order to make sure no block is skipped and they all performed as expected. The unique labels are processed in two complementary counters to comply with class B variable criteria. The main principle is a symmetrical four-step change of the counter pair content (adding or subtracting the unique label values) each time any significant testing block is processed. Two of the steps check if the block is correctly called from the main flow level (processed just before calling and just after return from the called procedure). The next two steps check if the block is correctly completed inside the called procedure (processed just after entry and just before return from the procedure). An example is given in *Figure 2*, where a routine performing a component test is called in the checked flow sequence and the four-step checking service is shown. This method decreases the load on the CPU as all these points are always checked by counting one member of the complementary counter pair only. As there is always the same number of call/return and entry/exit points, the values stored in the counter pair must be always complementary ones after any block is passed completely. Several execution flow check points are evaluated and placed in the code flow where the integrity of the counter pair is checked. If the counters are not complementary or they do not contain the expected values at any of these checkpoints, the Fail Safe routine is called.

Figure 2. Control flow four-step checking principle



Note: The unique number for the calling point for Component test 1 at the main level is defined as 5, and for the procedure itself it is defined as 7 in this example. The initial value of the counters are set to 0 and 0xFFFFFFFF for simplicity. The table in the upper right corner of Figure 2 shows how the counters are changed in four steps and their complementary state after the last step of the checking policy (return from procedure) is done.

2.2 Package organization

This section describes how the ST solution is organized.

2.2.1 Projects included in the package

As well as the standard firmware and self-test libraries, the installation package includes two projects for STM32 family - IAR's EWARM and Keil's MDK-ARM. The corresponding Project.eww or Project.uvproj project file must be configured for a specific STM32F10x family device before compilation.

The following STM32F10x members are supported:

- Low, medium, high and XL density Performance and Access line devices
- Low and medium density Value line devices

The required linker files are predefined and the defined symbols for concrete configuration are already declared in the preprocessor sections. For information on certain exceptions and special cases, refer to *Chapter 2.2.2: Tool-specific integration of the library*.

2.2.2 Tool-specific integration of the library

Startup file

It is recommended to reuse the following compiler-specific files:

- startup_stm32f10x_xd.s for IAR and startup_stm32f10x_xd.s for MDK-ARM
- The regular reset handler should be replaced by the STL_StartUp function address
- stm32f10x_flash_xd.icf (IAR) and BOOT_FLASH_XD_ClassB.sct (KEIL), linker script files where Class A / Class B variables regions are defined
- The workspace can also be used as an example (CRC generation is enabled in the linker/ processing tab of project options with IAR).
- Self-test startup routines do not alter or disable the compiler's standard C startup files, variables and stack/heap which are initialized in the usual way.

Defining new variables and memory sizes

Double storage in CLASS_B_RAM and CLASS_B_RAM_REV is necessary to ensure the redundancy of the safety critical data (Class B). All other variables defined without any particular attributes are considered as Class A variables and are not checked during the transparent RAM test.

Class A and Class B variable region sizes can be modified in the linker configuration file. New Class B variables must be declared in the ***stm32f10x_STLclassBvar.h*** header file, with the following syntax for the IAR development environment:

__no_init EXTERN uint32_t MyClassBvar @ "CLASS_B_RAM";
__no_init EXTERN uint32_t MyClassBvarInv @ "CLASS_B_RAM_REV";

and with the following syntax for the KEIL development environment:

EXTERN uint32_t MyClassBvar __attribute__((section("CLASS_B_RAM"), zero_init));
EXTERN uint32_t MyClassBvar Inv __attribute__((section("CLASS_B_RAM_REV"), zero_init));

Note: The start and end addresses of RAM/ROM regions are not exported when using the KEIL environment. These address modifications must be edited in the stm32f10x_STLparam.h file, which contains addresses and constants required to do the RAM and ROM (Flash) tests.

2.2.3 Application demo example

A short demo example of a user application is attached in the project main.c file (see *Chapter 3: Class B solution structure*). It provides an example of how the Class B routines can be integrated into an application-specific solution.

The demo example has been developed for the following evaluation boards:

- STM3210B-EVAL
- STM3210E-EVAL
- STM32100B-EVAL

The demo software uses the following hardware:

- USART1 Tx port (sending text messages for display in the hyper terminal window)
- PB9 Input for push-button scan (for debug purposes only)
- PC7 output to drive the LD2 LED indicating that software interrupt service routines are executed properly. The LED is toggled once the partial RAM test is completed (only the class B part of the RAM is tested).

2.3 Package configuration and debugging

Sometimes a functional part of the package should be suspended, excluded or included. The reason could be that certain tests are not required for a given application context. Some modifications can also useful when the package has to be debugged. This section describes how the ST solution can be configured, modified and debugged.

2.3.1 Configuration control

Configuration of the software is done at two basic levels. One configuration level is given by the project settings, where the differences between the STM32 family members is the factor. This part is mainly done automatically by proper configuration of the project. The other configuration level is the user settings. All user configuration settings are centralised in the Class B configuration file ***stm32f10x_STLparam.h***. A set of constants defined in this file control the conditional compilation of some functions. Adjustable constants must have specific settings to run all the tests properly (see note at *Chapter 2.2.1: Projects included in the package*).

Some runtime tests can be skipped depending on the end application. If the periodicity of the test is inherent (for example, for a washing machine, users switch the application on or off when it is used), then power on tests are sufficient and transparent/runtime tests can be avoided. This point must be discussed with the chosen test institute case by case.

For maximum robustness, the recommendation is to enable the independent watchdog using the hardware option bytes, and start the window watchdog as soon as possible in the main routine once the application development is close to the end. This is not done by default in the STM32 self-test library demo.

Stack overflow detection and watchdog self-check are not mandatory according to the EN/IEC60335-1 standard. They can be disabled or skipped if necessary.

It can help decrease the CPU load during runtime if 32-bit CRC checks are made using the STM32's internal CRC generator (32-bit wide CRC computation using the standard 0x04C11DB7 polynomial). Today, neither IAR nor KEIL support this 32-bit CRC format, but it can be used in conjunction with 16-bit CRC. One possible method could be to compute a 32-bit checksum in the initial phase and validate it using 16-bit CRC computation compared with the linker result. The validated 32-bit CRC value can be then saved as a reference for comparing all the subsequent runtime checks.

CRC generation is not supported in the KEIL environment, so calling CRC checking routines will cause the application to reset continuously (as if a failure was detected).

Examples:

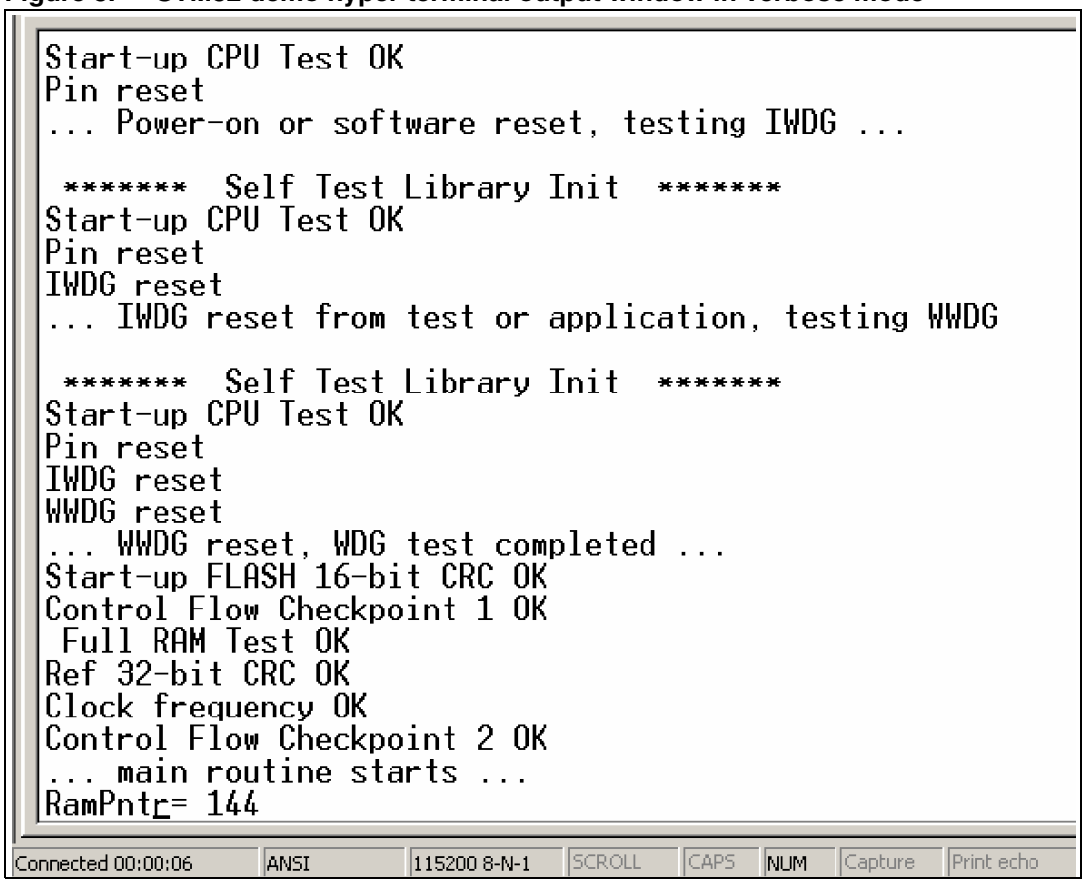
1. To disable CRC check at startup, modify the output logic control used to evaluate the test result (assuming the computed CRC is different from REF_CRC16):
   In the stm32f10x_STLstartup.c file modify line 173:
   *if(STL_crc16(CRC_INIT,(u8 *)ROM_START, ROM_SIZE) != REF_CRC16)*
   into:
   *if(STL_crc16(CRC_INIT,(u8 *)ROM_START, ROM_SIZE) == REF_CRC16)*
2. To replace 16-CRC by 32-bit internal CRC computation, modify stm32f10x_STLmain.c file line 207:
   RomTest = STL_crc16Run();
   by
   RomTest = STL_crc32Run();

Then comment the following #defines in file stm32f10x_STLparam.h starting from line 129:
#define DELTA_MAIN ....
#define LAST_DELTA_MAIN ....
#define FULL_FLASH_CHECKED ...
and uncomment the corresponding #defines for 32-bit CRC computation. In this case only the CRC32 algorithm control flow will be checked correctly.

2.3.2 Verbose diagnostic mode

The USART1 Tx serial peripheral line is used in verbose mode as a standard output for Class B status text messages. This mode is useful in the debug phase when the line can be monitored by an external terminal (the line setting is 115200 Bd, no parity, 8 bit data, 1 stop bit). Verbose mode is enabled by default and can be disabled during startup and/or runtime by commenting lines with the STL_VERBOSE_POR and STL_VERBOSE defined in the Class B configuration file ***stm32x_stl_param.h***. The pointer used for transparent RAM tests is displayed during the demo (RamPntr in *Figure 3*). A message is also displayed upon Flash CRC test completion ("Run-time FLASH CRC OK").

Figure 3. STM32 demo hyper terminal output window in verbose mode



2.3.3 Debugging the package

If any of the self-test routines fail, an MCU reset is triggered in the FailSafePOR function of the stm32f10x_STLstartup.c file. This makes the debugging of the application difficult, and can cause the debugger to lose the execution context.

While debugging the package it is useful to disable:

- The call to the NVIC_GenerateSystemReset() macro in the FailSafePOR() routine to prevent losing execution context when resetting the micro
- Control flow monitoring when adding or removing self-test routines, in particular run-time self-diagnostics
- All program memory CRC checksum tests when using software breaks in the code to prevent program memory check sum error occurrence
- Window watch dog to prevent improper service out of the time slot window dedicated for its refresh.

In the debugging phase it may be useful to enable the verbose diagnostic mode to watch Class B status text messages via the USART1 terminal.

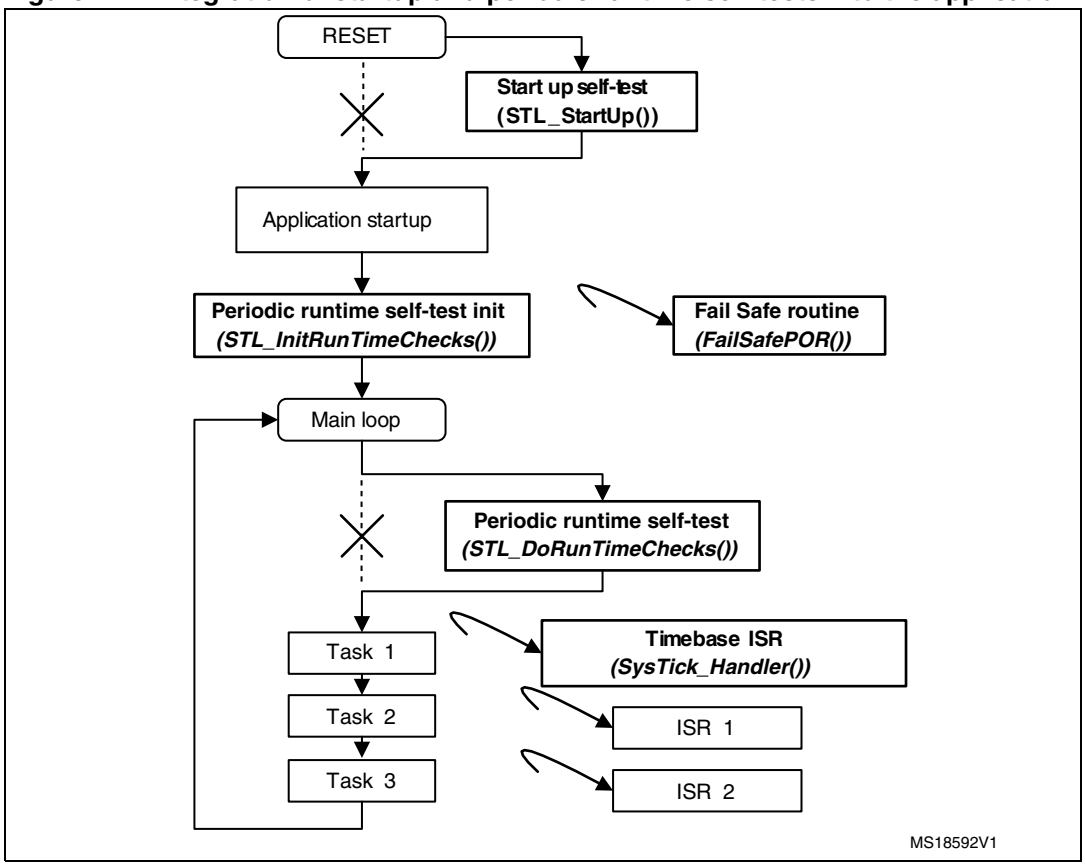
3 Class B solution structure

3.1 Integrating the software into the user application

Class B routines are divided into two main processes: startup and periodic runtime self tests. The periodic runtime test must be initialized by a set-up block before it is applied. All the processes are covered by adequate caller-called controls checked at a number of flow check points. All class B variables are kept redundantly in duplicate control registers stored in user-defined Class B variable space. This space is split into two separate RAM regions which are under permanent control of a transparent test which is part of the runtime tests.

Figure 4 shows the basic principle of how to integrate the Class B software package into a user software solution. The reset vector should be forced by the user to jump to the ***STL_StartUp()*** procedure which contains all the system self tests to be performed at startup. If all these tests pass sucessfully then the standard startup procedure ***__iar_program_start()*** for IAR or ***Reset_Handler()*** for the KEIL solution is performed.

Figure 4. Integration of startup and periodic runtime self-tests into the application



While the application is running, periodic tests are performed at certain intervals. To run them, users must execute an initialization block by calling ***STL_InitRunTimeChecks()*** routine in the initialization phase and then insert a periodic call to ***STL_DoRunTimeChecks()*** at the main level, preferably in the main loop. Periodic system interrupts are configured by initializing the SysTick clock frequency and themain time base measurement in the initial routine. The RTC is also reset and synchronized to allow

measurement to be performed in the SysTick interrupt service routine. The ***SysTick_Handler()*** interrupt service counts ticks and performs short partial transparent RAM March C or March X checks at correct intervals (20 ms) when a flag (*TimeBaseFlag*) is set to synchronize the rest of the runtime checks called from the main level. The ***FailSafePOR()*** routine itself is described in *Section 2.1.1: Fail Safe mode*.

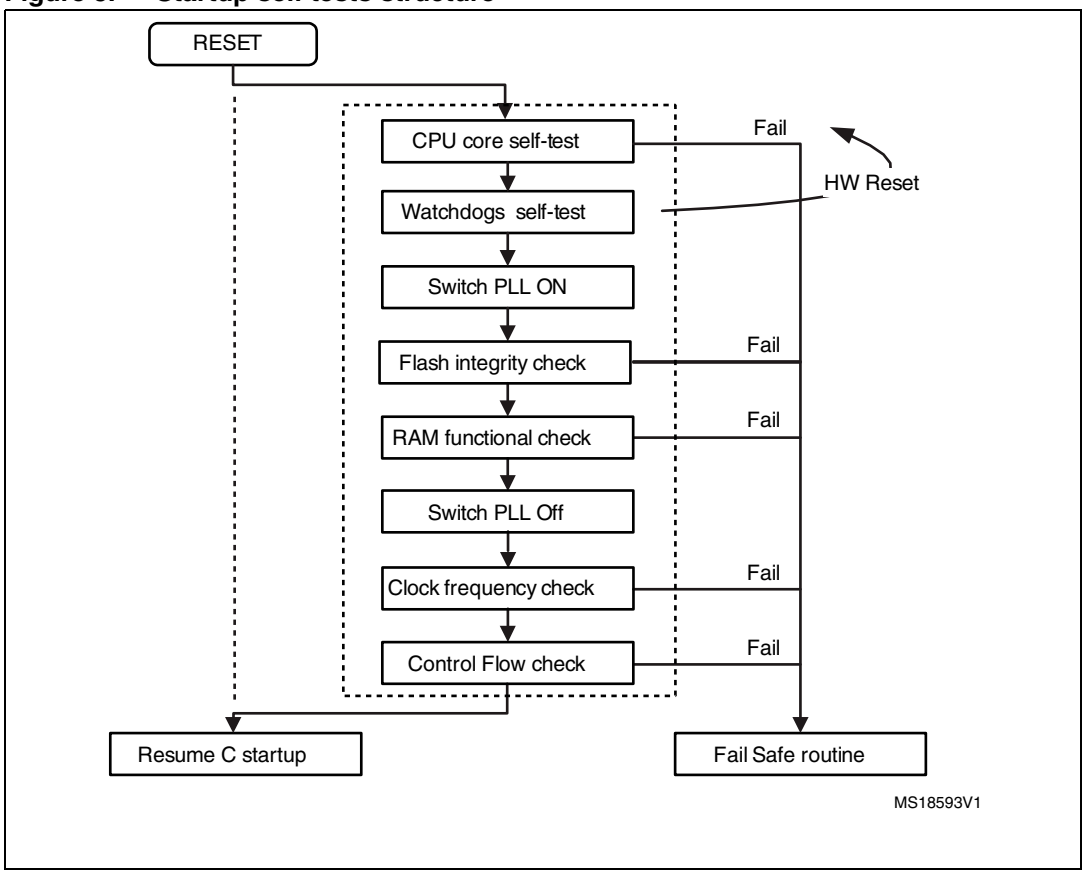
3.2 Description of startup self tests

The startup self-tests should be run during the initialization phase as the first check performed after resetting the micro (see *Figure 4*). This is done by forcing the reset vector to the address the start of the ***STL_StartUp()*** routine. The structure of the block of startup tests structure is shown in *Figure 5* and it includes the following self tests:

- CPU startup test
- Watchdog startup tests
- Flash complete checksum test
- Full RAM March C/X test
- Clock startup test
- Control flow check

These blocks are described in more detail further in this section.

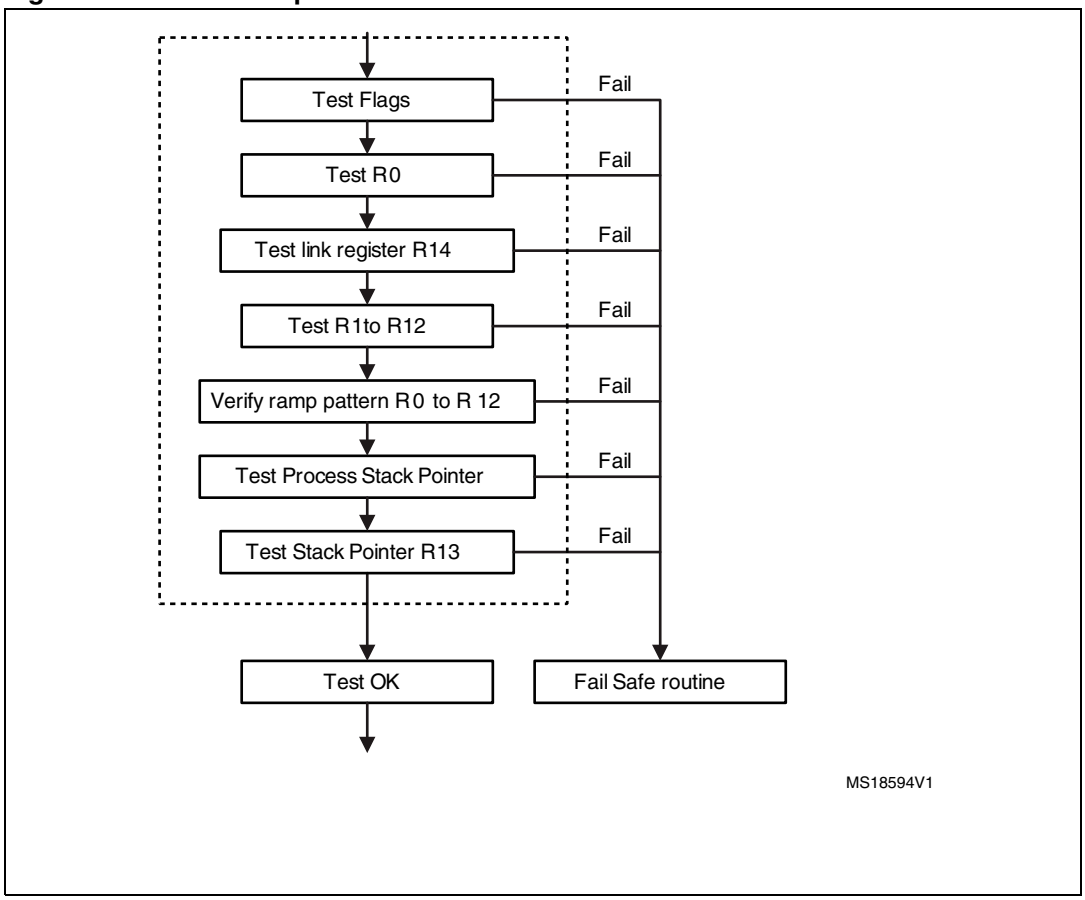
Figure 5. Startup self tests structure



3.2.1 CPU startup self-test

The CPU startup self-test tests the core flags, registers and stack pointers for correct functionality. If any error is found, it jumps directly to the Fail Safe routine. The source files are written in assembly , and there are two different files for the IAR and KEIL development environments. The structure is given in *Figure 6*.

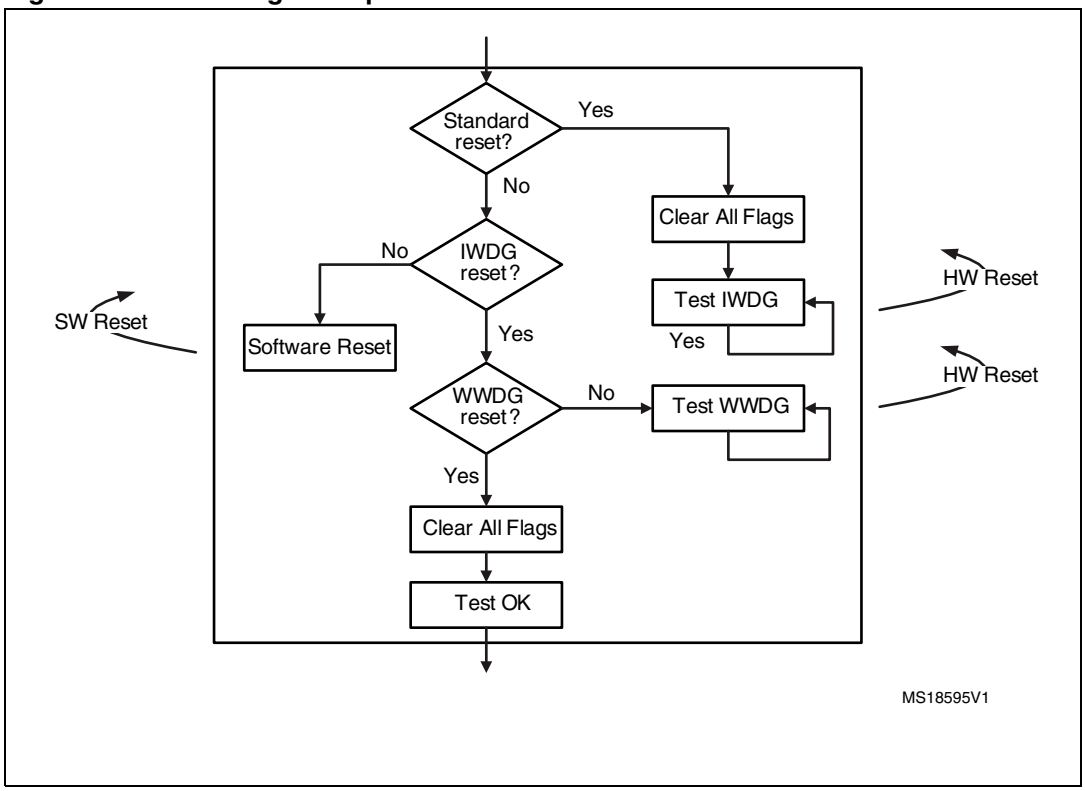
Figure 6. CPU startup self-test structure



3.2.2 Watch dog startup self-test

The structure of the test is based on the reset status register content which registers all the previous reset causes by setting different flags (see *Figure 7*).

Figure 7. Watchdog startup self-test structure



The standard reset condition (power on, low power, software or external pin flag indicates the previous reset cause) is assumed at the beginning of the watchdog test. All the flags are cleared, the IWDG timeout is set to the shortest period so an IWDG reset should then occur. After the next reset, the IWDG flag should be set and recognized as the sole reset cause. The test can then continue with the WWDG test. When both flags are set in reset status register, the test is considered as completed and all the flags in the reset status register are cleared again.

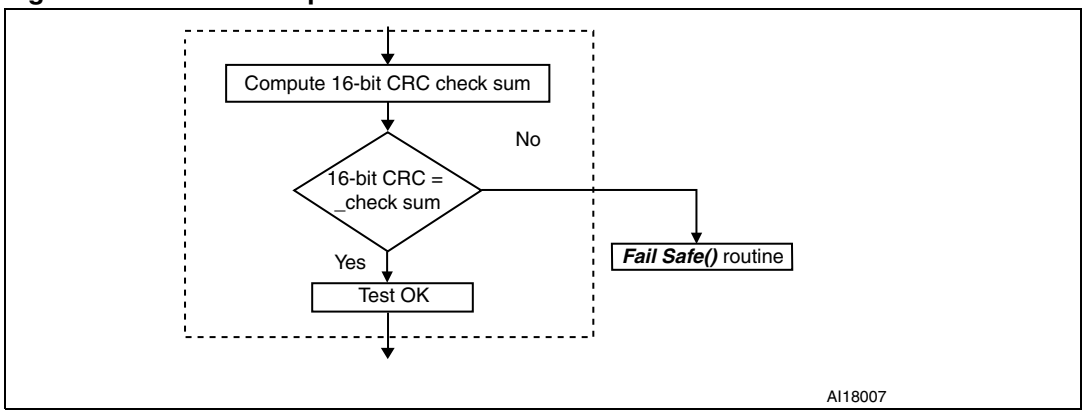
Users must take care to set both IWDG and WWDG periods properly. Their periods and the refresh window parameter must be set in accordance with the time base interval[(a)] because a normal refresh is performed at the successful end of the periodic runtime test in the main loop.

3.2.3 Flash complete checksum self-test

The CRC checksum computation is performed on the entire Flash space defined in the linker checksum structure. The result of the computation is compared with that of the linker. If they differ, the test fails. Refer to *Section 2.3.3: Debugging the package* for additional comments on CRC procedures.

a. The system tick interrupt service routine indicates a defined time base interval via a dedicated time base flag. The run-time test is started at the main level periodically at this interval. As the watchdog control is the last step of successfully finished run-time tests (and it should be the only place where the watchdog is refreshed in the main loop) the time base interval must be set in correlation with the watchdog timeout and vice versa. The watchdog refresh period must be shorter than the watchdog timeout to prevent a reset of the CPU. Refer to the flowchart in *Figure 14*.

Figure 8. Flash startup self-test structure



3.2.4 Full RAM March C-/X self-test

The entire RAM space is alternately checked and filled word by word with background patterns (value 0x00000000) and inverse background patterns (value 0xFFFFFFFF) in six loops as shown in *Figure 9*. The first three loops are performed in incremental order of addresses, and the last three in decremented reverse order. The order of tested addresses is scrambled as it respects the physical order of addresses to prevent and recognize any cross-talk between physically adjacent memory cells. The scramble principle is shown in *Table 3*. The basic physical unit is a pattern (a row) covering a block of 16 words. The numbers in the table cells represent logical addresses, while their order in the table represents the physical layout. Bold frames highlight the places where the logical order is scrambled.

Table 3. Physical order of RAM addresses organized into blocks of 16 words

	Physical order of addresses---->															
Rows -->	*0*	*1*	***3***	***2***	*4*	*5*	***7***	***6***	*8*	*9*	***11***	***10***	*12*	*13*	***15***	***14***
	16	*17*	***19***	***18***	*20*	*21*	***23***	***22***	*24*	*25*	***27***	***26***	*28*	*29*	***31***	***30***
	32	*33*	***35***	***34***	*36*	*37*	***39***	***38***	*40*	*...*						

The algorithm of the entire test and a detailed description one of its scramble loops is given in *Figure 9* and *Figure 10*. If an error is detected, the test is interrupted and a fail result is returned. The simplified March X algorithm can also be used. It is faster, as two middle marching steps are skipped.

Figure 9. RAM startup self-test structure

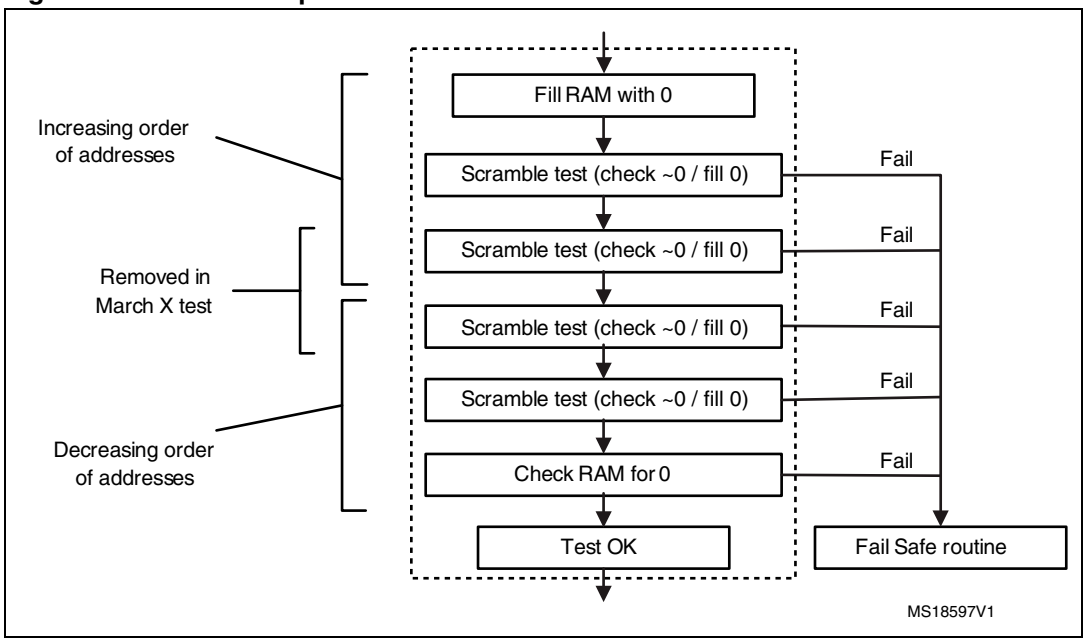
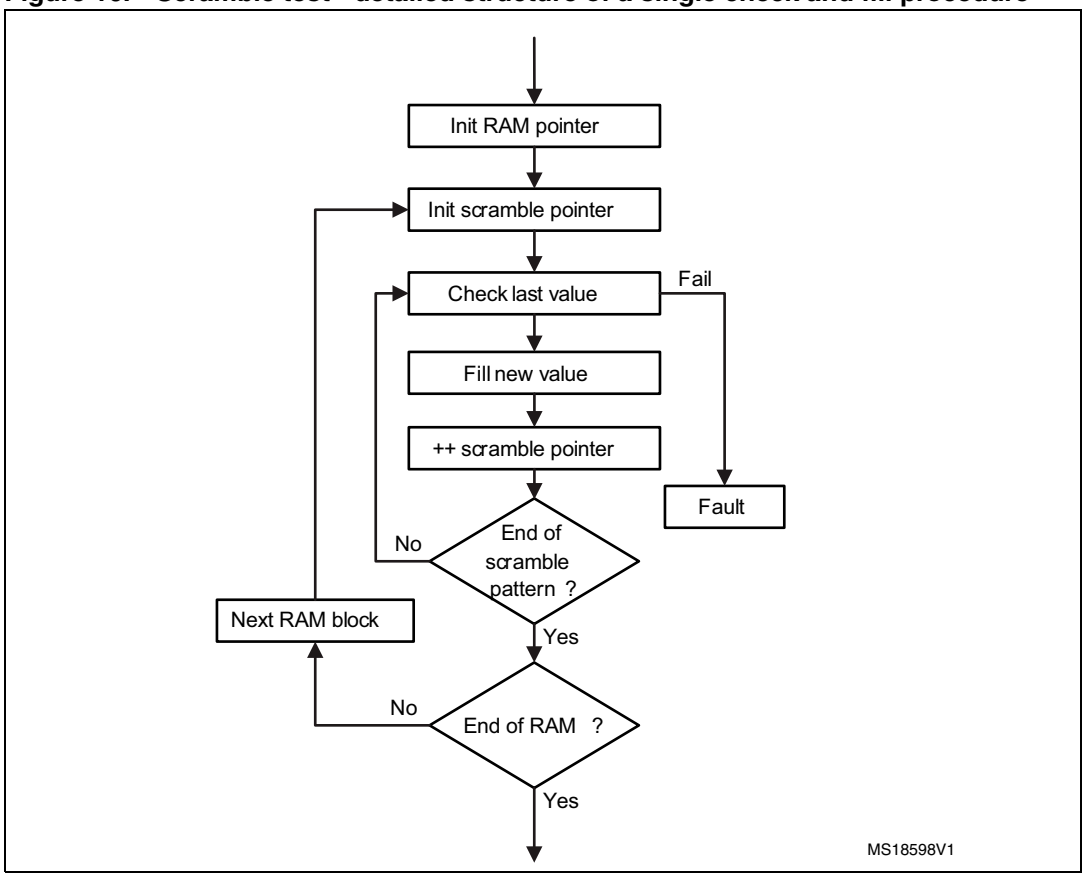


Figure 10. Scramble test - detailed structure of a single check and fill procedure

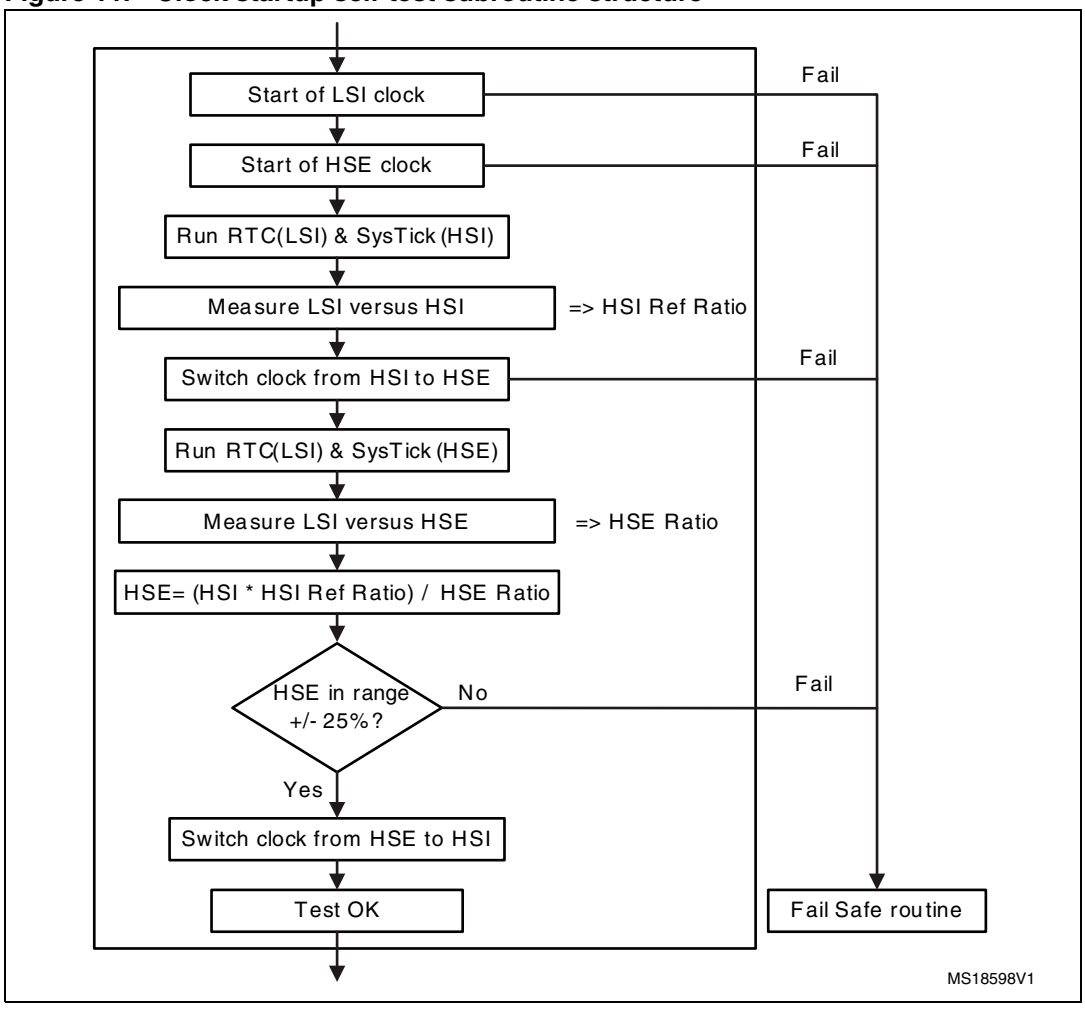


Note: The RAM test is byte oriented but the more precise bit March testing algorithm can be used, too. This method is more time and code consuming.

3.2.5 Clock startup self-test

The flow chart diagram of the test is shown in *Figure 11*. It expresses the principle of the method, graphically. Initially the internal low speed clock (LSI) source is started. The external high speed source (HSE) is started in the next step. The CPU is still running on the internal high speed clock source (HSI). The system timer (SysTick) is started with the HSI clock source while the RTC system runs on the LSI clock source. One complete underflow cycle of the SysTick timer is performed. The number of LSI periods counted in that cycle gives the HSI ratio value (number of HSI pulses within the interval corresponding to one LSI period). This initial HSI ratio measurement is stored as a reference for the next runtime measurements.

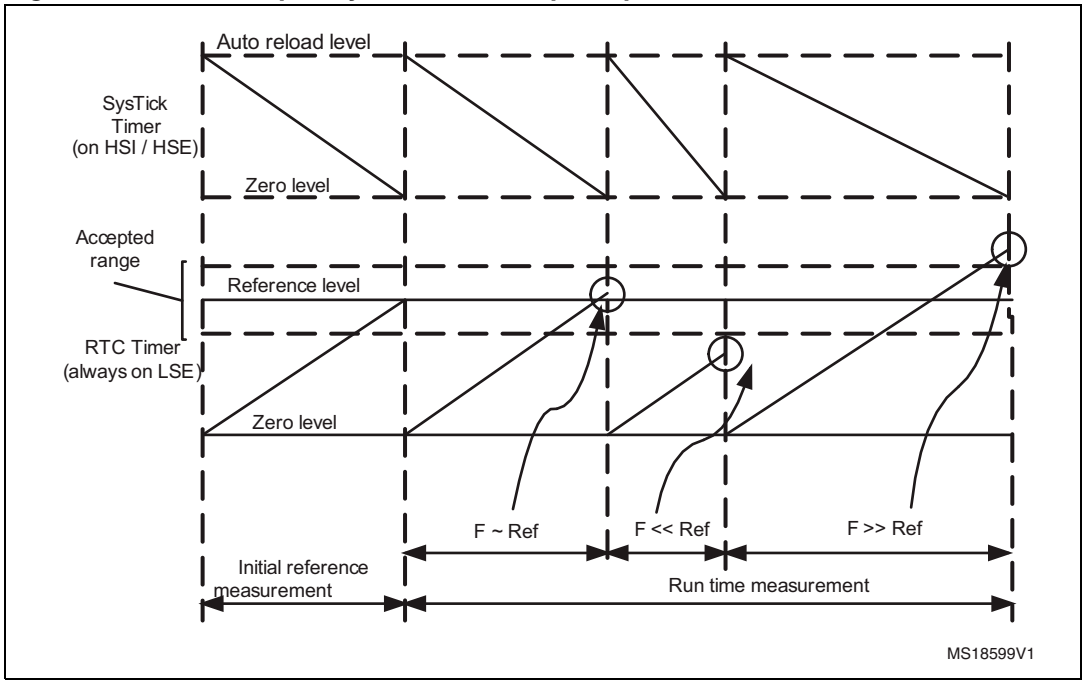
Figure 11. Clock startup self-test subroutine structure



In the next step, the external HSE clock is switched as the new clock source for the SysTick timer. Another complete underflow cycle of SysTick timer is performed. The number of LSI periods counted in this cycle gives the HSE ratio (number of HSE pulses in the interval corresponding to one LSI period). The HSE value is then computed from the known HSI value and both HSI and HSE ratios measured. If the HSE value deviates from the expected interval (more than +/- 25% of its nominal value), the CPU clock source is immediately switched back to HSI, and HSE fail status is returned. Otherwise the test returns the OK

status. In either case, the CPU clock is switched back to the default HSI source after the test is finished.

Figure 12. Clock frequency measurement principle

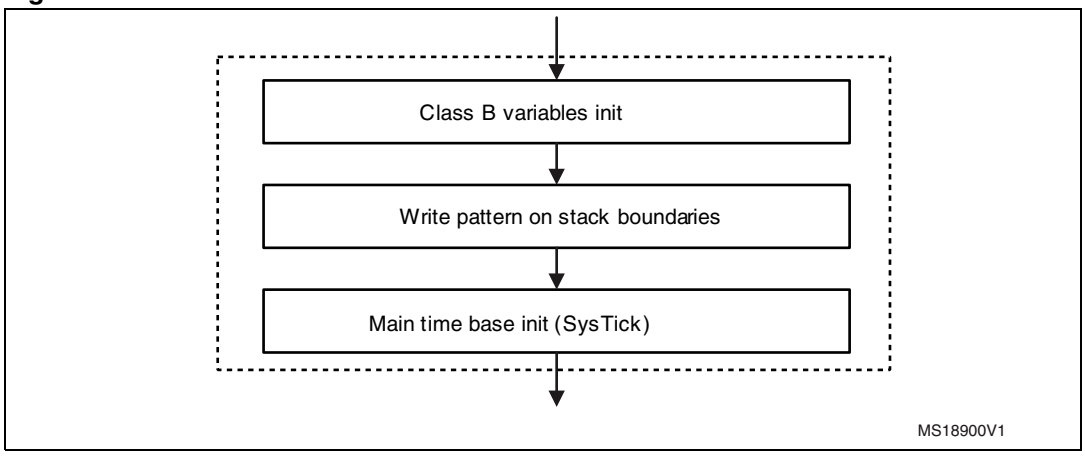


3.3 Periodic runtime self-test initialization

Assuming that the startup self-tests passed sucessfully and standard initialization has been done, then the runtime self-tests package must be initialized just before the program enters the main loop performing regular calling of the runtime self-tests (refer to *Figure 4*). The timing should be set properly to ensure the procedures of the runtime tests are called at the necessary intervals.

At first all class B variables are initialized. Zero and its complement value are stored into every class B variable complementary pair. The magic pattern is then stored at the top of the space separated for the stack. Timer peripherals are configured for the tick interval measurement and master clock frequency measurement. The same method as the startup test is used.

Figure 13. Periodic runtime self-test initialization structure



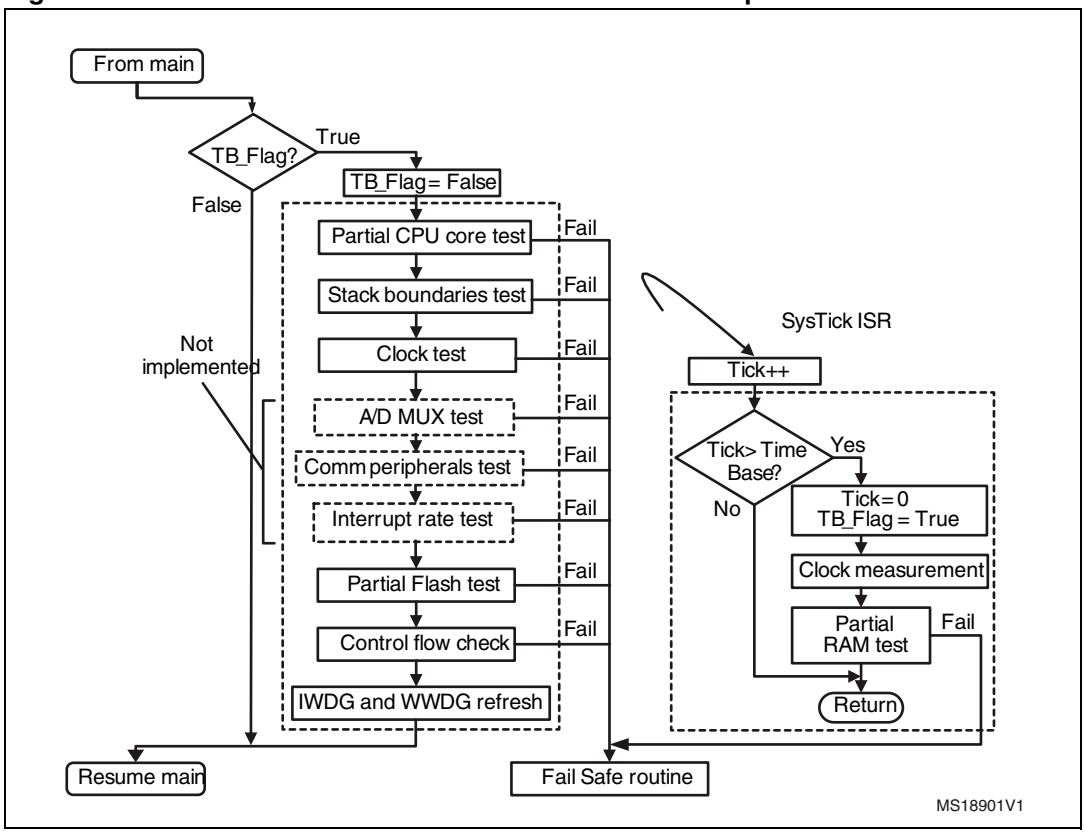
3.4 Description of periodic runtime self-tests

3.4.1 Runtime self-test structure

The Runtime self-test is a block of tests which is performed periodically at the main loop level. The execution period of the block is based on timebase interrupt settings. Before the first run, all the tests included must be initialized by the runtime initialization phase block (refer to *Figure 4*). Most of these tests are performed at the main loop level. Only the partial transparent RAM test and the backup of the latest clock measurement results are performed in the SysTick timer interrupt service routine at 20 ms intervals, signaled by TimeBaseFlag settings. It includes the following self-tests:

- CPU core partial runtime test
- Stack boundaries overflow test
- Clock runtime test
- AD MUX self-test (not implemented)
- Interrupt rate test (not implemented)
- Communication peripherals test (not implemented)
- Flash partial CRC test including evaluation of the complete test
- Independent and window watchdog refresh
- Partial transparent RAM March C-/X test (under system interrupt scope)

Figure 14. Periodic runtime self-test and timebase interrupt service structure

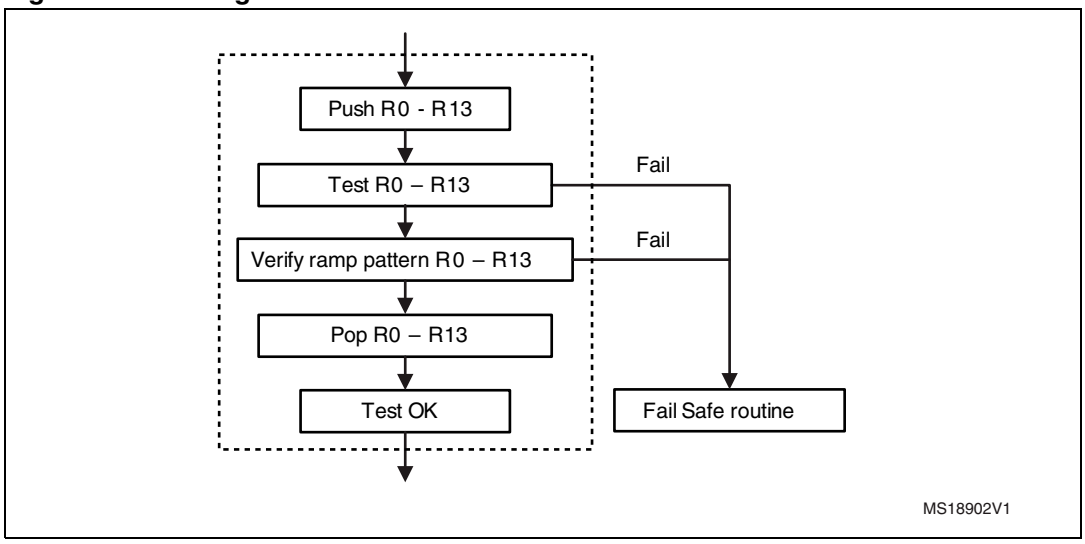


Note: Tests of the analog part, communication peripherals and application interrupts are not included and their implementation depends on the specific microcontroller device capabilities and user application needs.

3.4.2 CPU light runtime self-test

The light runtime CPU core self-tests is a simplified version of the runtime test described in *Section 3.4.1: Runtime self-test structure*. The flags and stack pointers are not tested. If an error code is returned the Fail Save procedure is called.

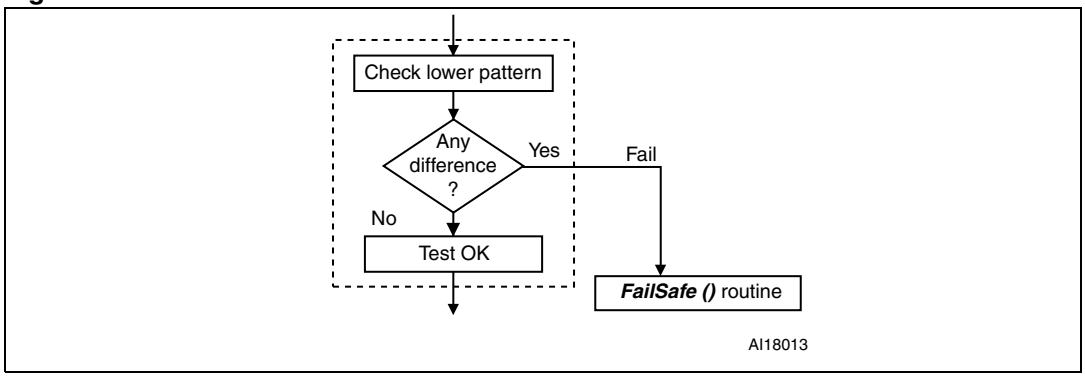
Figure 15. CPU light runtime self-test structure



3.4.3 Stack boundaries runtime test

This test checks the magic pattern stored at the top of the space reserved for the stack. If the original pattern is corrupted, the Fail Safe routine is called. The pattern is placed at the lowest address reserved for the stack area. It can detect both overflow and underflow as the stack pointer rolls over this range in either case. The stack area differs depending on the specific microcontroller device type. Be careful of the stack area limits when the location of the pattern is changed.

Figure 16. Stack overflow runtime test structure

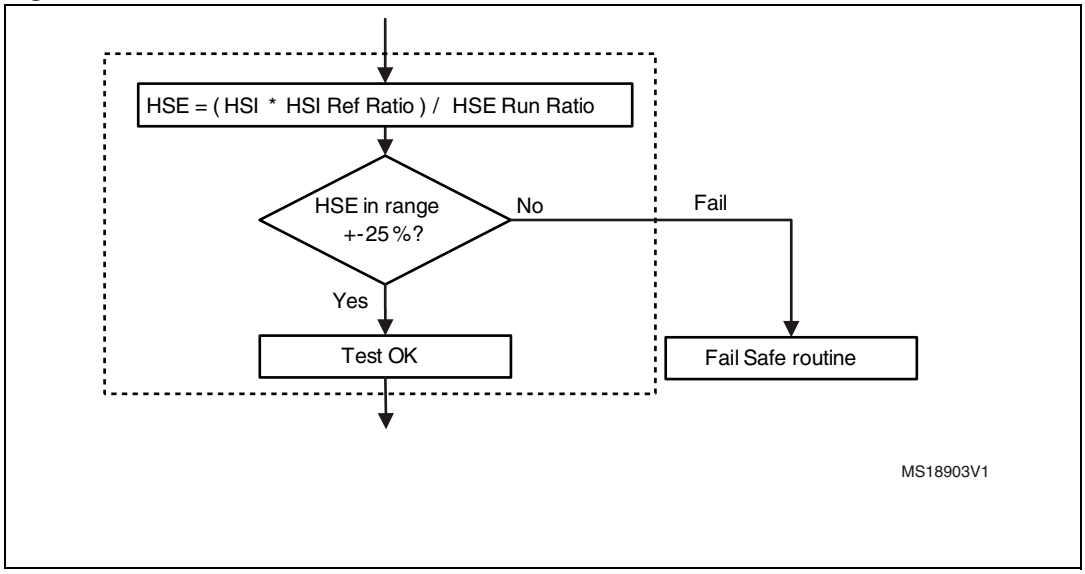


3.4.4 Clock runtime self-test

The clock runtime self-test uses a similar procedure to the one used in the startup self-test. A few underflow cycles of the SysTick timer are performed at background processing level. The number of LSI periods counted during this period by the RTC timer gives the runtime HSE ratio (number of HSE pulses within the interval corresponding to the LSI period) which is stored as a Class B variable by the STL_MeasurePeriod() function called by the SysTick interrupt service routine. The HSE value is then computed at the main processing level in the clock run self-test. The inputs are the same as in the startup test: known HSI reference ratio value (stored in the initial startup test phase) and the currently measured HSE ratio. If the computed HSE value deviates from the expected interval (more than +/- 25% of its

nominal value) the CPU clock source is immediately switched back to to HSI, and HSE fail status is returned. Otherwise the test returns OK status.

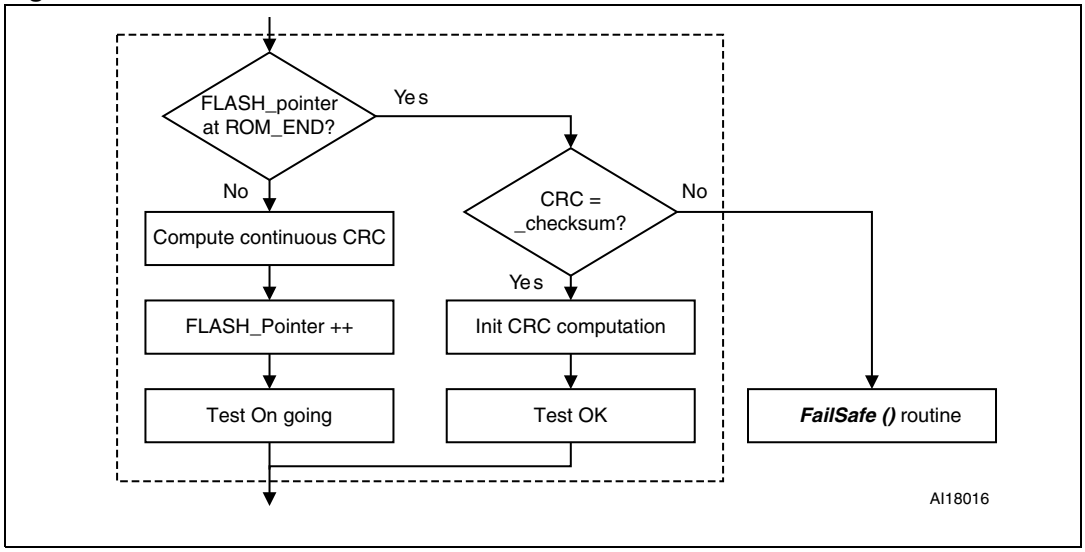
Figure 17. Clock runtime self-test structure



3.4.5 Partial Flash CRC runtime self-test

The partial 16-bit CRC checksum of the block in Flash is performed at each step. The boundaries are given by the structure stored by the linker. When the last block is reached, the CRC checksum is compared with the value stored by the linker. If a difference is found, the Fail Safe routine is called, otherwise a new computation cycle is initialized. Refer also to *Section 2.3.3: Debugging the package* for additional comments on CRC procedures.

Figure 18. Partial Flash CRC runtime self-test structure



3.4.6 Watchdog service in runtime test

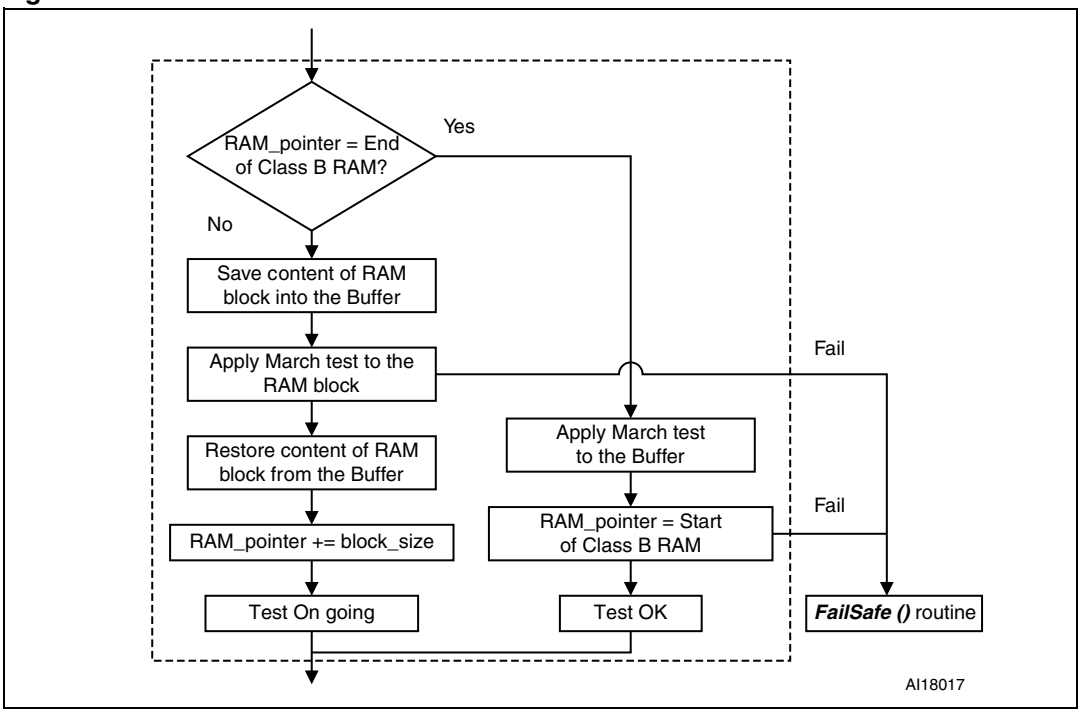
If the runtime service block is successfully completed, the window and independent watchdogs should both be refreshed as a last step, just before returning to the main loop. For the watchdogs to be refreshed correctly, proper timing of the call to the runtime block is essential. The period when the block should be called is signaled internally by a time base flag which is tested at the beginning of the ***STL_DoRunTimeChecks()*** routine (see *Figure 14*). Users must ensure not to miss calling this procedure at the main level to be able to react properly to any change in the time base flag and consequently refresh the watchdogs at the correct intervals.

To use the watchdogs efficiently, it is important to keep the structure of the application with only one refresh placed in the main loop. There should be no other watchdog refreshes except the one in the ***STL_DoRunTimeChecks()*** routine. Sometimes it may also be necessary to refresh watchdogs in the initialization phase of the flow. In this case, the refresh should be outside any software infinite loop. Ideally it should only be put in a straightforward part of the code.

3.4.7 Partial RAM runtime self-test

The partial transparent RAM test is performed inside the timebase interrupt service routine. The test covers the part of the RAM allocated to class B variables. One block of six bytes is tested in each step of the test. To guarantee coupling fault coverage, the blocks of memory are overlapped by two side bytes from one test step to the other. The order of testing the bytes in the block is scrambled with respect to the physical order of addresses in memory. For more details see *Section 3.2.4: Full RAM March C-/X self-test*.

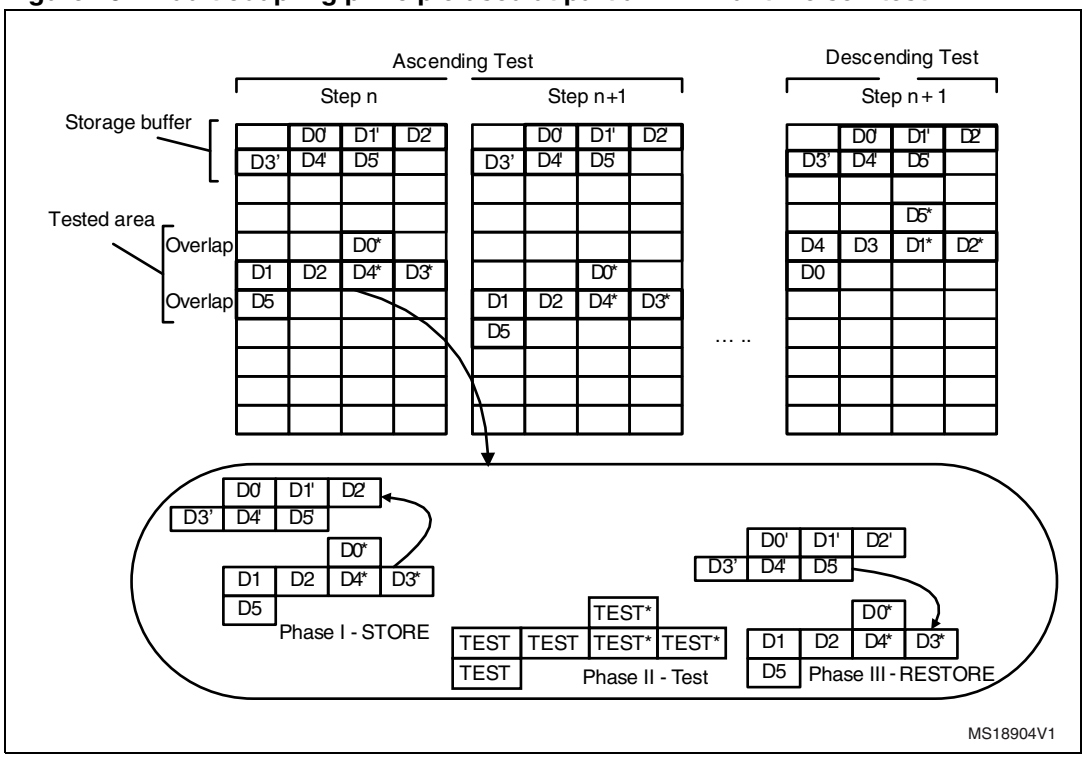
Figure 19. Partial RAM runtime self-test structure



In the first phase, the content of the block is stored in the storage buffer. Marching destructive tests are performed over all the bytes of the tested RAM block in the next phase. Then, in the final phase of this step, the original content is restored from the storage buffer.

The March X algorithm is faster because two middle marching steps are skipped. As a last step of the test sequence, the storage buffer itself is tested by a marching test. The buffer is tested together with the next two additional bytes to cover coupling faults in the buffer itself. Then the whole test is reinitialized and it starts from the beginning. If any fault is detected, the Fail Safe routine is called.

Figure 20. Fault coupling principle used at partial RAM runtime self-test



Note: Scrambling of physical addresses is respected in both ascending and descending tests.

Table 4. March C- phases in RAM partial test

March phase	Partial bytes test over the block	Address order
Initial	Write 0x00 pattern	Increasing
1	Test 0x00 pattern, write 0xFF pattern	Increasing
2	Test 0xFF pattern, write 0x00 pattern	Increasing
3	Test 0x00 pattern, write 0xFF pattern	Decreasing
4	Test 0xFF pattern, write 0x00 pattern	Decreasing
5	Test 0x00 pattern	Decreasing

Note: Steps 2 and 3 are skipped when the March X algorithm is used.

Appendix A Library reference

Table 5. List of self-test library files

File name	Description	CA[1]
stm32f10x_STLclassBvar.h	Class B variable definitions	X
stm32f10x_STLclockrun.c	Crystal / resonator frequency check runtime procedures	
stm32f10x_STLclockrun.h		
stm32f10x_STLclockstart.c	Crystal / resonator frequency check startup procedures	
stm32f10x_STLclockstart.h		
stm32f10x_STLcpu.h	Cortex-M3 core self-diagnostic runtime procedures	
stm32f10x_STLcpurunIAR.s		
stm32f10x_STLcpurunKEIL.s		
stm32f10x_STLcpustartIAR.s	Cortex-M3 core self-diagnostic startup procedures	
stm32f10x_STLcpustartKEIL.s		
stm32f10x_STLcrc16.c	16-bit CRC full FLASH check startup procedures	
stm32f10x_STLcrc16.h		
stm32f10x_STLcrc16Run.c	16-bit CRC block FLASH check runtime procedures	
stm32f10x_STLcrc16Run.h		
stm32f10x_STLcrc32.c	32-bit CRC full FLASH check startup procedures	
stm32f10x_STLcrc32.h		
stm32f10x_STLcrc32Run.c	32-bit CRC block FLASH check runtime procedures	
stm32f10x_STLcrc32Run.h		
stm32f10x_STLfullRamMc.c	Full RAM functional startup test using March C-	
stm32f10x_STLfullRamMc.h		
stm32f10x_STLlib.h	All STM32 self-test library header files #include	
stm32f10x_STLmain.c	All the self-test routines calls to be executed at runtime except SysTick interrupt service[2]	X
stm32f10x_STLmain.h		X
stm32f10x_STLparam.h	STM32 self-test library parameters	X
stm32f10x_STLstartup.c	All the self-test routines calls to be executed at startup	
stm32f10x_STLstartup.h		
stm32f10x_STLtranspRamMc.c	Partial transparent RAM functional runtime test using March C-	
stm32f10x_STLtranspRamMc.h		
stm32f10x_STLtranspRamMx.c	Partial transparent RAM functional runtime test using March X	
stm32f10x_STLtranspRamMx.c		

1. Procedures to be customized
2. SysTick_Handler is a part of stm32f10x_it.c file

4 Revision history

Table 6. Revision history

Date	Revision	Description of changes
09-Dec-2010	1	Initial release
27-May-2011	2	Added the following text to cover page: The associated firmware package is available upon request only. Please contact your local ST sales office for more information.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2011 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com